13 | 第一个C函数：如何实现板级初始化？

2021-06-07 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨
时长 21:11 大小 19.41M

你好，我是 LMOS。

前面三节课，我们为调用 Cosmos 的**第一个 C 函数 hal_start 做了大量工作。**这节课我们要让操作系统 Cosmos 里的第一个 C 函数真正跑起来啦，也就是说，我们会真正进入到我们的内核中。

今天我们会继续在这个 hal_start 函数里，首先执行板级初始化，其实就是 hal 层（硬件抽象层，下同）初始化，其中执行了平台初始化，hal 层的内存初始化，中断初始化，最后进入到内核层的初始化。

这节课的配套代码，你可以从这里下载。

第一个 C 函数

任何软件工程，第一个函数总是简单的，因为它是总调用者，像是一个管理者，坐在那里发号施令，自己却是啥活也不干。

由于这是第一个 C 函数，也是初始化函数，我们还是要为它单独建立一个文件，以显示对它的尊重，依然在 Cosmos/hal/x86/ 下建立一个 hal_start.c 文件。写上这样一个函数。

复制代码

```
void hal_start()
{
    //第一步：初始化hal层
    //第二步：初始化内核层
    for(;;);
    return;
}
```

根据前面的设计，Cosmos 是有 hal 层和内核层之分，所以在上述代码中，要分两步走。第一步是初始化 hal 层；第二步，初始化内核层。只是这两步的函数我们还没有写。

然而最后的死循环却有点奇怪，其实它的目的很简单，就是避免这个函数返回，因为这个返回了就无处可去，避免走回头路。

hal 层初始化

为了分离硬件的特性，我们设计了 hal 层，把硬件相关的操作集中在这个层，并向上提供接口，目的是让内核上层不用关注硬件相关的细节，也能方便以后移植和扩展。(关于 hal 层的设计，可以回顾第 3 节课)

也许今天我们是在 x86 平台上写 Cosmos，明天就要在 ARM 平台上开发 Cosmos，那时我们就可以写个 ARM 平台的 hal 层，来替换 Cosmos 中的 x86 平台的 hal 层。

下面我们在 Cosmos/hal/x86/ 下建立一个 halinit.c 文件，写出 hal 层的初始化函数。

复制代码

```
void init_hal()
{
    //初始化平台
```

```
    //初始化内存
    //初始化中断
    return;
}
```

这个函数也是一个调用者，没怎么干活。不过根据代码的注释能看出，它调用的函数多一点，但主要是完成初始化平台、初始化内存、初始化中断的功能函数。

初始化平台

我们先来写好平台初始化函数，因为它需要最先被调用。

这个函数主要负责完成两个任务，一是**把二级引导器建立的机器信息结构复制到 hal 层中的一个全局变量中**，方便内核中的其它代码使用里面的信息，之后二级引导器建立的数据所占用的内存都会被释放。二是要**初始化图形显示驱动**，内核在运行过程要在屏幕上输出信息。

下面我们在 Cosmos/hal/x86/ 下建立一个 halplatform.c 文件，写上如下代码。

复制代码

```
void machbstart_t_init(machbstart_t *initp)
{
    //清零
    memset(initp, 0, sizeof(machbstart_t));
    return;
}

void init_machbstart()
{
    machbstart_t *kmbsp = &kmachbsp;
    machbstart_t *smbsp = MBSPADR;//物理地址1MB处
    machbstart_t_init(kmbsp);
    //复制，要把地址转换成虚拟地址
    memcopy((void *)phyadr_to_viradr((adr_t)smbsp), (void *)kmbsp, sizeof(mach
    return;
}
//平台初始化函数
void init_halplaltform()
{
    //复制机器信息结构
    init_machbstart();
    //初始化图形显示驱动
    init_bdvideo();
```

```
    return;
}
```

这个代码中别的地方很好理解，就是 kmachbsp 你可能会有点奇怪，它是个结构体变量，结构体类型是 machbstart_t，这个结构和二级引导器所使用的一模一样。

同时，它还是一个 hal 层的全局变量，我们想专门有个文件定义所有 hal 层的全局变量，于是我们在 Cosmos/hal/x86/ 下建立一个 halglobal.c 文件，写上如下代码。

复制代码

```
//全局变量定义变量放在data段
#define HAL_DEFGLOB_VARIABLE(vartype,varname) \
EXTERN  __attribute__((section(".data"))) vartype varname

HAL_DEFGLOB_VARIABLE(machbstart_t,kmachbsp);
```

前面的 EXTERN，在 halglobal.c 文件中定义为空，而在其它文件中定义为 extern，告诉编译器这是外部文件的变量，避免发生错误。

下面，我们在 Cosmos/hal/x86/ 下的 bdvideo.c 文件中，写好 init_bdvideo 函数。

复制代码

```
void init_bdvideo()
{
    dftgraph_t *kghp = &kdftgh;
    //初始化图形数据结构，里面放有图形模式，分辨率，图形驱动函数指针
    init_dftgraph();
    //初始bga图形显卡的函数指针
    init_bga();
    //初始vbe图形显卡的函数指针
    init_vbe();
    //清空屏幕 为黑色
    fill_graph(kghp, BGRA(0, 0, 0));
    //显示背景图片
    set_charsdxwflush(0, 0);
    hal_background();
    return;
}
```

init_defgraph() 函数初始了 dftgraph_t 结构体类型的变量 kdftgh，我们在 halglobal.c 文件中定义这个变量，结构类型我们这样来定义。

复制代码

```
typedef struct s_DFTGRAPH
{
    u64_t gh_mode;          //图形模式
    u64_t gh_x;             //水平像素点
    u64_t gh_y;             //垂直像素点
    u64_t gh_framphyadr;    //显存物理地址
    u64_t gh_fvrmphyadr;    //显存虚拟地址
    u64_t gh_fvrmsz;        //显存大小
    u64_t gh_onepixbits;    //一个像素字占用的数据位数
    u64_t gh_onepixbyte;
    u64_t gh_vbemodenr;     //vbe模式号
    u64_t gh_bank;          //显存的bank数
    u64_t gh_curdipbnk;     //当前bank
    u64_t gh_nextbnk;       //下一个bank
    u64_t gh_banksz;        //bank大小
    u64_t gh_fontadr;       //字库地址
    u64_t gh_fontsz;        //字库大小
    u64_t gh_fnthight;      //字体高度
    u64_t gh_nxtcharsx;     //下一字符显示的x坐标
    u64_t gh_nxtcharsy;     //下一字符显示的y坐标
    u64_t gh_linesz;        //字符行高
    pixl_t gh_deffontpx;    //默认字体大小
    u64_t gh_chardxw;
    u64_t gh_flush;
    u64_t gh_framnr;
    u64_t gh_fshdata;       //刷新相关的
    dftghops_t gh_opfun;    //图形驱动操作函数指针结构体
}dftgraph_t;
typedef struct s_DFTGHOPS
{
    //读写显存数据
    size_t (*dgo_read)(void* ghpdev,void* outp,size_t rdsz);
    size_t (*dgo_write)(void* ghpdev,void* inp,size_t wesz);
    sint_t (*dgo_ioctrl)(void* ghpdev,void* outp,uint_t iocode);
    //刷新
    void   (*dgo_flush)(void* ghpdev);
    sint_t (*dgo_set_bank)(void* ghpdev, sint_t bnr);
    //读写像素
    pixl_t (*dgo_readpix)(void* ghpdev,uint_t x,uint_t y);
    void   (*dgo_writepix)(void* ghpdev,pixl_t pix,uint_t x,uint_t y);
    //直接读写像素
    pixl_t (*dgo_dxreadpix)(void* ghpdev,uint_t x,uint_t y);
    void   (*dgo_dxwritepix)(void* ghpdev,pixl_t pix,uint_t x,uint_t y);
    //设置x，y坐标和偏移
    sint_t (*dgo_set_xy)(void* ghpdev,uint_t x,uint_t y);
    sint_t (*dgo_set_vwh)(void* ghpdev,uint_t vwt,uint_t vhi);
```

```
    sint_t (*dgo_set_xyoffset)(void* ghpdev,uint_t xoff,uint_t yoff);
    //获取x，y坐标和偏移
    sint_t (*dgo_get_xy)(void* ghpdev,uint_t* rx,uint_t* ry);
    sint_t (*dgo_get_vwh)(void* ghpdev,uint_t* rvwt,uint_t* rvhi);
    sint_t (*dgo_get_xyoffset)(void* ghpdev,uint_t* rxoff,uint_t* ryoff);
}dftghops_t;
//刷新显存
void flush_videoram(dftgraph_t *kghp)
{
    kghp->gh_opfun.dgo_flush(kghp);
    return;
}
```

不难发现，我们正是把这些实际的图形驱动函数的地址填入了这个结构体中，然后通过这个结构体，我们就可以调用到相应的函数了。

因为写这些函数都是体力活，我已经帮你搞定了，你直接使用就可以。上面的 flush_videoram 函数已经证明了这一想法。

来，我们测试一下，看看结果，我们图形驱动程序初始化会显示背景图片——background.bmp，这是在打包映像文件时包含进去的，你自己可以随时替换，只要是满足 **1024*768，24 位的位图文件**就行了。

下面我们要把这些函数调用起来：

复制代码

```
//在halinit.c文件中
void init_hal()
{
    init_halplaltform();
    return;
}
//在hal_start.c文件中
void hal_start()
{
    init_hal();//初始化hal层，其中会调用初始化平台函数，在那里会调用初始化图形驱动
    for(;;);
    return;
}
```

接下来，让我们一起 make vboxtest，应该很有成就感。一幅风景图呈现在我们面前，上面有 Cosmos 的版本、编译时间、CPU 工作模式，内存大小等数据。这相当一个我们 Cosmos 的水印信息。

图形驱动测试

初始化内存

首先，我们在 Cosmos/hal/x86/ 下建立一个 halmm.c 文件，用于初始化内存，为了后面的内存管理器作好准备。

hal 层的内存初始化比较容易，只要向内存管理器提供内存空间布局信息就可以。

你可能在想，不对啊，明明我们在二级引导器中已经获取了内存布局信息，是的，**但 Cosmos 的内存管理器需要保存更多的信息，最好是顺序的内存布局信息，这样可以增加**

额外的功能属性，同时降低代码的复杂度。

不难发现，BIOS 提供的结构无法满足前面这些要求。不过我们也有办法解决，只要以 BIOS 提供的结构为基础，设计一套新的数据结构就搞定了。这个结构可以这样设计。

复制代码

```
#define PMR_T_OSAPUSERRAM 1
#define PMR_T_RESERVRAM 2
#define PMR_T_HWUSERRAM 8
#define PMR_T_ARACONRAM 0xf
#define PMR_T_BUGRAM 0xff
#define PMR_F_X86_32 (1<<0)
#define PMR_F_X86_64 (1<<1)
#define PMR_F_ARM_32 (1<<2)
#define PMR_F_ARM_64 (1<<3)
#define PMR_F_HAL_MASK 0xff

typedef struct s_PHYMMARGE
{
    spinlock_t pmr_lock;//保护这个结构是自旋锁
    u32_t pmr_type;     //内存地址空间类型
    u32_t pmr_stype;
    u32_t pmr_dtype;    //内存地址空间的子类型，见上面的宏
    u32_t pmr_flgs;     //结构的标志与状态
    u32_t pmr_stus;
    u64_t pmr_saddr;    //内存空间的开始地址
    u64_t pmr_lsize;    //内存空间的大小
    u64_t pmr_end;      //内存空间的结束地址
    u64_t pmr_rrvmsaddr;//内存保留空间的开始地址
    u64_t pmr_rrvmend;  //内存保留空间的结束地址
    void* pmr_prip;     //结构的私有数据指针，以后扩展所用
    void* pmr_extp;     //结构的扩展数据指针，以后扩展所用
}phymmarge_t;
```

有些情况下内核要另起炉灶，不想把所有的内存空间都交给内存管理器去管理，所以要保留一部分内存空间，这就是上面结构中那两个 pmr_rrvmsaddr、pmr_rrvmend 字段的作用。

有了数据结构，我们还要写代码来操作它：

复制代码

```
u64_t initpmrge_core(e820map_t *e8sp, u64_t e8nr, phymmarge_t *pmargesp)
{
```

```
    u64_t retnr = 0;
    for (u64_t i = 0; i < e8nr; i++)
    {
        //根据一个e820map_t结构建立一个phymmarge_t结构
        if (init_one_pmrge(&e8sp[i], &pmargesp[i]) == FALSE)
        {
            return retnr;
        }
        retnr++;
    }
    return retnr;
}
void init_phymmarge()
{
    machbstart_t *mbsp = &kmachbsp;
    phymmarge_t *pmarge_adr = NULL;
    u64_t pmrgesz = 0;
    //根据machbstart_t机器信息结构计算获得phymmarge_t结构的开始地址和大小
    ret_phymmarge_adrandsz(mbsp, &pmarge_adr, &pmrgesz);
    u64_t tmppmrphyadr = mbsp->mb_nextwtpadr;
    e820map_t *e8p = (e820map_t *)((adr_t)(mbsp->mb_e820padr));
    //建立phymmarge_t结构
    u64_t ipmgnr = initpmrge_core(e8p, mbsp->mb_e820nr, pmarge_adr);
    //把phymmarge_t结构的地址大小个数保存machbstart_t机器信息结构中
    mbsp->mb_e820expadr = tmppmrphyadr;
    mbsp->mb_e820exnr = ipmgnr;
    mbsp->mb_e820exsz = ipmgnr * sizeof(phymmarge_t);
    mbsp->mb_nextwtpadr = PAGE_ALIGN(mbsp->mb_e820expadr + mbsp->mb_e820exsz);
    //phymmarge_t结构中地址空间从低到高进行排序，我已经帮你写好了
    phymmarge_sort(pmarge_adr, ipmgnr);
    return;
}
```

结合上面的代码，你会发现这是根据 e820map_t 结构数组，建立了一个 phymmarge_t 结构数组，init_one_pmrge 函数正是把 e820map_t 结构中的信息复制到 phymmarge_t 结构中来。理解了这个原理，即使不看我的，你自己也会写。

下面我们把这些函数，用一个总管函数调动起来，这个总管函数叫什么名字好呢？当然是 init_halmm，如下所示。

复制代码

```
void init_halmm()
{
    init_phymmarge();
    //init_memmgr();
    return;
}
```

这里 init_halmm 函数中还调用了 init_memmgr 函数，这个正是这我们内存管理器初始化函数，我会在内存管理的那节课展开讲。而 init_halmm 函数将要被 init_hal 函数调用。

初始化中断

什么是中断呢？为了帮你快速理解，我们先来看两种情景：

1. 你在开车时，突然汽车引擎坏了，你需要修复它才能继续驾驶汽车……
2. 你在外旅游，你女朋友突然来电话了，你可以选择接电话或者不接电话，当然不接电话的后果很严重（笑）……

在以上两种情景中，虽然不十分恰当，但都是在做一件事时，因为一些原因而要切换到另一件事上。其实计算机中的 CPU 也是一样，在做一件事时，因为一些原因要转而做另一件事，于是中断产生了……

根据原因的类型不同，中断被分为两类。

异常，这是同步的，原因是错误和故障，就像汽车引擎坏了。不修复错误就不能继续运行，所以这时，CPU 会跳到这种错误的处理代码那里开始运行，运行完了会返回。

为啥说它是同步的呢？这是因为如果不修改程序中的错误，下次运行程序到这里同样会发生异常。

中断，这是异步的，我们通常说的中断就是这种类型，它是因为外部事件而产生的，就好像旅游时女朋友来电话了。通常设备需要 CPU 关注时，会给 CPU 发送一个中断信号，所以这时 CPU 会跳到处理这种事件的代码那里开始运行，运行完了会返回。

由于不确定何种设备何时发出这种中断信号，所以它是异步的。

在 x86 CPU 上，最多支持 256 个中断，还记得前面所说的中断表和中断门描述符吗，这意味着我们要准备 256 个中断门描述符和 256 个中断处理程序的入口。

下面我们来定义它，如下所示：

复制代码

```
typedef struct s_GATE
{
        u16_t   offset_low;     /* 偏移 */
        u16_t   selector;       /* 段选择子 */
        u8_t    dcount;         /* 该字段只在调用门描述符中有效。如果在利用调用门调用子
        u8_t    attr;           /* P(1) DPL(2) DT(1) TYPE(4) */
        u16_t   offset_high;    /* 偏移的高位段 */
        u32_t   offset_high_h;
        u32_t   offset_resv;
}__attribute__((packed)) gate_t;
//定义中断表
HAL_DEFGLOB_VARIABLE(gate_t,x64_idt)[IDTMAX];
```

说到这里你会发现，中断表其实是个 gate_t 结构的数组，由 CPU 的 IDTR 寄存器指向，IDTMAX 为 256。

但是光有数组还不行，还要设置其中的数据，下面我们就来设计这个函数，建立一个文件 halsgdidt.c，在其中写一个函数，代码如下。

复制代码

```
//vector 向量也是中断号
//desc_type 中断门类型，中断门，陷阱门
//handler 中断处理程序的入口地址
//privilege 中断门的权限级别
void set_idt_desc(u8_t vector, u8_t desc_type, inthandler_t handler, u8_t priv
{
    gate_t *p_gate = &x64_idt[vector];
    u64_t base = (u64_t)handler;
    p_gate->offset_low = base & 0xFFFF;
    p_gate->selector = SELECTOR_KERNEL_CS;
    p_gate->dcount = 0;
    p_gate->attr = (u8_t)(desc_type | (privilege << 5));
    p_gate->offset_high = (u16_t)((base >> 16) & 0xFFFF);
    p_gate->offset_high_h = (u32_t)((base >> 32) & 0xffffffff);
    p_gate->offset_resv = 0;
    return;
}
```

上面的代码，正是按照要求，把这些数据填入中断门描述符中的。有了中断门之后，还差中断处理程序，中断处理程序只负责这三件事：

1. 保护 CPU 寄存器，即中断发生时的程序运行的上下文。

2. 调用中断处理程序，这个程序可以是修复异常的，可以是设备驱动程序中对设备响应的程序。

3. 恢复 CPU 寄存器，即恢复中断时程序运行的上下文，使程序继续运行。

以上这些操作又要用汇编代码才可以编写，我觉得这是内核中最重要的部分，所以我们建立一个文件，并用 kernel.asm 命名。

我们先来写好完成以上三个功能的汇编宏代码，避免写 256 遍同样的代码，代码如下所示。

复制代码

```
1  //保存中断后的寄存器
2  %macro  SAVEALL  0
3    push rax
4    push rbx
5    push rcx
6    push rdx
7    push rbp
8    push rsi
9    push rdi
10   push r8
11   push r9
12   push r10
13   push r11
14   push r12
15   push r13
16   push r14
17   push r15
18   xor r14,r14
19   mov r14w,ds
20   push r14
21   mov r14w,es
22   push r14
23   mov r14w,fs
24   push r14
25   mov r14w,gs
26   push r14
```

```
%endmacro
//恢复中断后寄存器
%macro  RESTOREALL  0
  pop r14
  mov gs,r14w
  pop r14
  mov fs,r14w
  pop r14
  mov es,r14w
  pop r14
  mov ds,r14w
  pop r15
  pop r14
  pop r13
  pop r12
  pop r11
  pop r10
  pop r9
  pop r8
  pop rdi
  pop rsi
  pop rbp
  pop rdx
  pop rcx
  pop rbx
  pop rax
  iretq
%endmacro
//保存异常下的寄存器
%macro  SAVEALLFAULT 0
  push rax
  push rbx
  push rcx
  push rdx
  push rbp
  push rsi
  push rdi
  push r8
  push r9
  push r10
  push r11
  push r12
  push r13
  push r14
  push r15
  xor r14,r14
  mov r14w,ds
  push r14
  mov r14w,es
  push r14
  mov r14w,fs
  push r14
```

```
  mov r14w,gs
  push r14
%endmacro
//恢复异常下寄存器
%macro  RESTOREALLFAULT  0
  pop r14
  mov gs,r14w
  pop r14
  mov fs,r14w
  pop r14
  mov es,r14w
  pop r14
  mov ds,r14w
  pop r15
  pop r14
  pop r13
  pop r12
  pop r11
  pop r10
  pop r9
  pop r8
  pop rdi
  pop rsi
  pop rbp
  pop rdx
  pop rcx
  pop rbx
  pop rax
  add rsp,8
  iretq
%endmacro
//没有错误码CPU异常
%macro  SRFTFAULT 1
  push    _NOERRO_CODE
  SAVEALLFAULT
  mov r14w,0x10
  mov ds,r14w
  mov es,r14w
  mov fs,r14w
  mov gs,r14w
  mov   rdi,%1 ;rdi, rsi
  mov   rsi,rsp
  call   hal_fault_allocator
  RESTOREALLFAULT
%endmacro
//CPU异常
%macro  SRFTFAULT_ECODE 1
  SAVEALLFAULT
  mov r14w,0x10
  mov ds,r14w
  mov es,r14w
  mov fs,r14w
```

```
131   mov gs,r14w
132   mov   rdi,%1
133   mov   rsi,rsp
134   call   hal_fault_allocator
135   RESTOREALLFAULT
136 %endmacro
137 //硬件中断
138 %macro  HARWINT  1
139   SAVEALL
140   mov r14w,0x10
141   mov ds,r14w
142   mov es,r14w
143   mov fs,r14w
144   mov gs,r14w
145   mov  rdi, %1
146   mov   rsi,rsp
147   call    hal_intpt_allocator
148   RESTOREALL
149
```

别看前面的代码这么长，其实**最重要的只有两个指令：push、pop**，这两个正是用来压入寄存器和弹出寄存器的，正好可以用来保存和恢复 CPU 所有的通用寄存器。

有的 CPU 异常，CPU 自动把异常码压入到栈中，而有的 CPU 异常没有异常码，**为了统一，我们对没有异常码的手动压入一个常数，维持栈的平衡。**

有了中断异常处理的宏，我们还要它们变成中断异常的处理程序入口点函数。汇编函数其实就是一个标号加一段汇编代码，C 编译器把 C 语言函数编译成汇编代码后，也是标号加汇编代码，函数名就是标号。

下面我们在 kernel.asm 中写好它们：

复制代码

```
1  //除法错误异常 比如除0
2  exc_divide_error:
3    SRFTFAULT 0
4  //单步执行异常
5  exc_single_step_exception:
6    SRFTFAULT 1
7  exc_nmi:
8    SRFTFAULT 2
9  //调试断点异常
10 exc_breakpoint_exception:
11   SRFTFAULT 3
12 //溢出异常
```

```
exc_overflow:
  SRFTFAULT 4
//段不存在异常
exc_segment_not_present:
  SRFTFAULT_ECODE 11
//栈异常
exc_stack_exception:
  SRFTFAULT_ECODE 12
//通用异常
exc_general_protection:
  SRFTFAULT_ECODE 13
//缺页异常
exc_page_fault:
  SRFTFAULT_ECODE 14
hxi_exc_general_intpfault:
  SRFTFAULT 256
//硬件1～7号中断
hxi_hwint00:
  HARWINT  (INT_VECTOR_IRQ0+0)
hxi_hwint01:
  HARWINT  (INT_VECTOR_IRQ0+1)
hxi_hwint02:
  HARWINT  (INT_VECTOR_IRQ0+2)
hxi_hwint03:
  HARWINT  (INT_VECTOR_IRQ0+3)
hxi_hwint04:
  HARWINT  (INT_VECTOR_IRQ0+4)
hxi_hwint05:
  HARWINT  (INT_VECTOR_IRQ0+5)
hxi_hwint06:
  HARWINT  (INT_VECTOR_IRQ0+6)
hxi_hwint07:
  HARWINT  (INT_VECTOR_IRQ0+7)
```

为了突出重点，这里没有全部展示代码 ，你只用搞清原理就行了。那有了中断处理程序的入口地址，下面我们就可以在 halsgdidt.c 文件写出函数设置中断门描述符了，代码如下。

复制代码

```
void init_idt_descriptor()
{
//一开始把所有中断的处理程序设置为保留的通用处理程序
    for (u16_t intindx = 0; intindx <= 255; intindx++)
    {
        set_idt_desc((u8_t)intindx, DA_386IGate, hxi_exc_general_intpfault, PR
    }
    set_idt_desc(INT_VECTOR_DIVIDE, DA_386IGate, exc_divide_error, PRIVILEGE_K
    set_idt_desc(INT_VECTOR_DEBUG, DA_386IGate, exc_single_step_exception, PRI
    set_idt_desc(INT_VECTOR_NMI, DA_386IGate, exc_nmi, PRIVILEGE_KRNL);
```

```
    set_idt_desc(INT_VECTOR_BREAKPOINT, DA_386IGate, exc_breakpoint_exception,
    set_idt_desc(INT_VECTOR_OVERFLOW, DA_386IGate, exc_overflow, PRIVILEGE_USE
//篇幅所限，未全部展示
    set_idt_desc(INT_VECTOR_PAGE_FAULT, DA_386IGate, exc_page_fault, PRIVILEGE
    set_idt_desc(INT_VECTOR_IRQ0 + 0, DA_386IGate, hxi_hwint00, PRIVILEGE_KRNL
    set_idt_desc(INT_VECTOR_IRQ0 + 1, DA_386IGate, hxi_hwint01, PRIVILEGE_KRNL
    set_idt_desc(INT_VECTOR_IRQ0 + 2, DA_386IGate, hxi_hwint02, PRIVILEGE_KRNL
    set_idt_desc(INT_VECTOR_IRQ0 + 3, DA_386IGate, hxi_hwint03, PRIVILEGE_KRNL
    //篇幅所限，未全部展示
     return;
}
```

上面的代码已经很明显了，一开始把所有中断的处理程序设置为保留的通用处理程序，避免未知中断异常发生了 CPU 无处可去，然后对已知的中断和异常进一步设置，这会覆盖之前的通用处理程序，这样就可以确保万无一失。

下面我们把这些代码整理一下，安装到具体的调用路径上，让上层调用者调用到就好了。

我们依然在 halintupt.c 文件中写上 init_halintupt() 函数：

复制代码

```
void init_halintupt()
{
    init_idt_descriptor();
    init_intfltdsc();
    return;
}
```

到此为止，CPU 体系层面的中断就初始化完成了。你会发现，我们在 init_halintupt() 函数中还调用了 **init_intfltdsc() 函数**，这个函数是干什么的呢？请往下看。

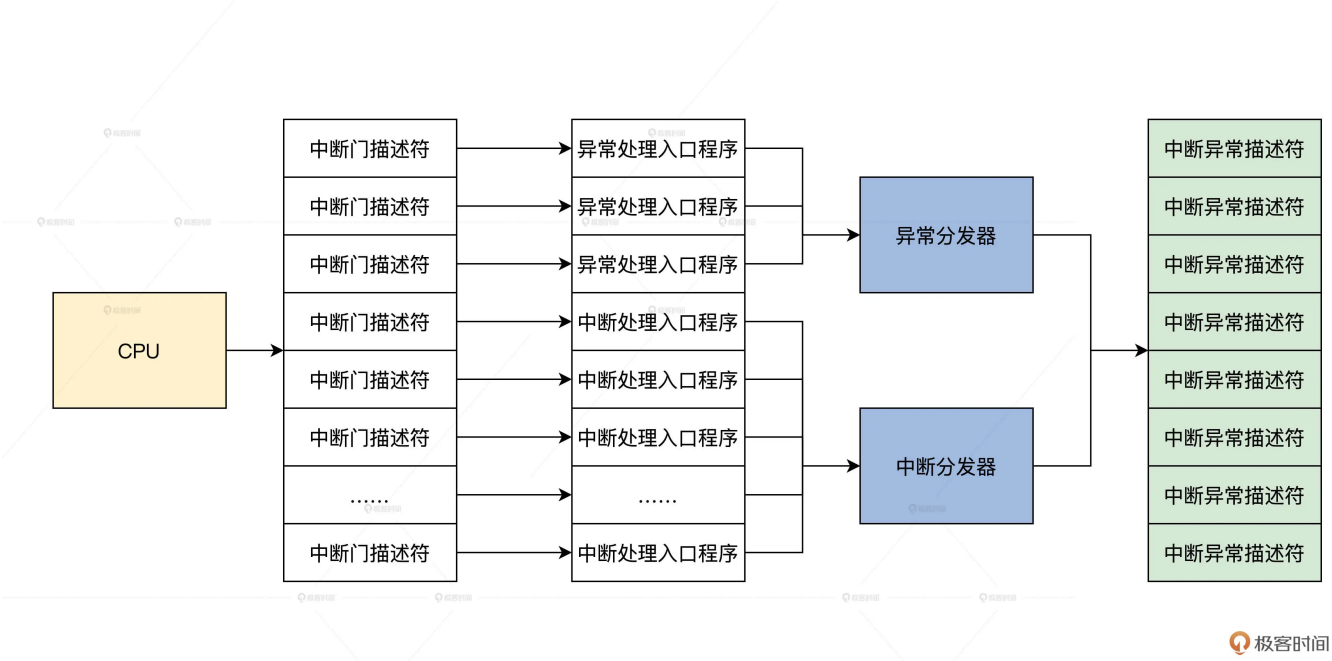
我们先来设计一下 Cosmos 的中断处理框架，后面我们把中断和异常统称为中断，因为它们的处理方式相同。

前面我们只是解决了中断的 CPU 相关部分，而 CPU 只是响应中断，但是并不能解决产生中断的问题。

比如缺页中断来了，我们要解决内存地址映射关系，程序才可以继续运行。再比如硬盘中断来了，我们要读取硬盘的数据，要处理这问题，就要写好相应的处理函数。

因为有些处理是内核所提供的，而有些处理函数是设备驱动提供的，想让它们和中断关联起来，就要好好设计**中断处理框架**了。

下面我们来画幅图，描述中断框架的设计：



中断框架设计图

可以看到，中断、异常分发器的左侧的东西我们已经处理完成，下面需要写好中断、异常分发器和中断异常描述符。

我们先来搞定中断异常描述，结合框架图，中断异常描述也是个表，它在 C 语言中就是个结构数组，让我们一起来写好这个数组：

复制代码

```
typedef struct s_INTFLTDSC{
    spinlock_t  i_lock;
    u32_t       i_flg;
    u32_t       i_stus;
    uint_t      i_prity;        //中断优先级
    uint_t      i_irqnr;        //中断号
    uint_t      i_deep;         //中断嵌套深度
    u64_t       i_indx;         //中断计数
    list_h_t    i_serlist;      //也可以使用中断回调函数的方式
    uint_t      i_sernr;        //中断回调函数个数
    list_h_t    i_serthrdlst;   //中断线程链表头
    uint_t      i_serthrdnr;    //中断线程个数
    void*       i_onethread;    //只有一个中断线程时直接用指针
    void*       i_rbtreeroot;   //如果中断线程太多则按优先级组成红黑树
```

```
    list_h_t    i_serfisrlst;
    uint_t      i_serfisrnr;
    void*       i_msgmpool;     //可能的中断消息池
    void*       i_privp;
    void*       i_extp;
}intfltdsc_t;
```

上面结构中，记录了中断了优先级。因为有些中断可以稍后执行，而有的中断需要紧急执行，所以要设计一个优先级。其中还有中断号，中断计数等统计信息。

中断可以由线程的方式执行，也可以是一个回调函数，该函数的地址放另一个结构体中，这个结构体我已经帮你写好了，如下所示。

复制代码

```
typedef drvstus_t (*intflthandle_t)(uint_t ift_nr,void* device,void* sframe);
typedef struct s_INTSERDSC{
    list_h_t    s_list;         //在中断异常描述符中的链表
    list_h_t    s_indevlst;     //在设备描述描述符中的链表
    u32_t       s_flg;
    intfltdsc_t* s_intfltp;     //指向中断异常描述符
    void*       s_device;       //指向设备描述符
    uint_t      s_indx;
    intflthandle_t s_handle;    //中断处理的回调函数指针
}intserdsc_t;
```

如果内核或者设备驱动程序要安装一个中断处理函数，就要先申请一个 intserdsc_t 结构体，然后把中断函数的地址写入其中，最后把这个结构挂载到对应的 intfltdsc_t 结构中的 i_serfisrlst 链表中。

你可能要问了，为什么不能直接把中断处理函数放在 intfltdsc_t 结构中呢，还要多此一举搞个 intserdsc_t 结构体呢？

这是因为我们的计算机中可能有很多设备，每个设备都可能产生中断，但是中断控制器的中断信号线是有限的。你可以这样理解：中断控制器最多只能产生几十号中断号，而设备不止几十个，所以会有多个设备共享一根中断信号线。

这就导致一个中断发生后，无法确定是哪个设备产生的中断，所以我们干脆让设备驱动程序来决定，因为它是最了解设备的。

这里我们让这个 intfltdsc_t 结构上的所有中断处理函数都依次执行，查看是不是自己的设备产生了中断，如果是就处理，不是则略过。

好，明白了这两个结构之后，我们就要开始初始化了。首先是在 halglobal.c 文件定义 intfltdsc_t 结构。

复制代码

```
//定义intfltdsc_t结构数组大小为256
HAL_DEFGLOB_VARIABLE(intfltdsc_t,machintflt)[IDTMAX];
```

下面我们再来实现中断、异常分发器函数，如下所示。

复制代码

```
//中断处理函数
void hal_do_hwint(uint_t intnumb, void *krnlsframp)
{
    intfltdsc_t *ifdscp = NULL;
    cpuflg_t cpuflg;
    //根据中断号获取中断异常描述符地址
    ifdscp = hal_retn_intfltdsc(intnumb);
    //对断异常描述符加锁并中断
    hal_spinlock_saveflg_cli(&ifdscp->i_lock, &cpuflg);
    ifdscp->i_indx++;
    ifdscp->i_deep++;
    //运行中断处理的回调函数
    hal_run_intflthandle(intnumb, krnlsframp);
    ifdscp->i_deep--;
    //解锁并恢复中断状态
    hal_spinunlock_restflg_sti(&ifdscp->i_lock, &cpuflg);
    return;
}
//异常分发器
void hal_fault_allocator(uint_t faultnumb, void *krnlsframp)
{
    //我们的异常处理回调函数也是放在中断异常描述符中的
    hal_do_hwint(faultnumb, krnlsframp);
    return;
}
//中断分发器
void hal_hwint_allocator(uint_t intnumb, void *krnlsframp)
{
    hal_do_hwint(intnumb, krnlsframp);
    return;
}
```

前面的代码确实是按照我们的中断框架设计实现的，下面我们去实现 hal_run_intflthandle 函数，它负责调用中断处理的回调函数。

复制代码

```
void hal_run_intflthandle(uint_t ifdnr, void *sframe)
{
    intserdsc_t *isdscp;
    list_h_t *lst;
    //根据中断号获取中断异常描述符地址
    intfltdsc_t *ifdscp = hal_retn_intfltdsc(ifdnr);
    //遍历i_serlist链表
    list_for_each(lst, &ifdscp->i_serlist)
    {
        //获取i_serlist链表上对象即intserdsc_t结构
        isdscp = list_entry(lst, intserdsc_t, s_list);
        //调用中断处理回调函数
        isdscp->s_handle(ifdnr, isdscp->s_device, sframe);
    }
    return;
}
```

上述代码已经很清楚了，循环遍历 intfltdsc_t 结构中，i_serlist 链表上所有挂载的 intserdsc_t 结构，然后调用 intserdsc_t 结构中的中断处理的回调函数。

我们 Cosmos 链表借用了 Linux 所用的链表，代码我已经帮你写好了，放在了 list.h 和 list_t.h 文件中，请自行查看。

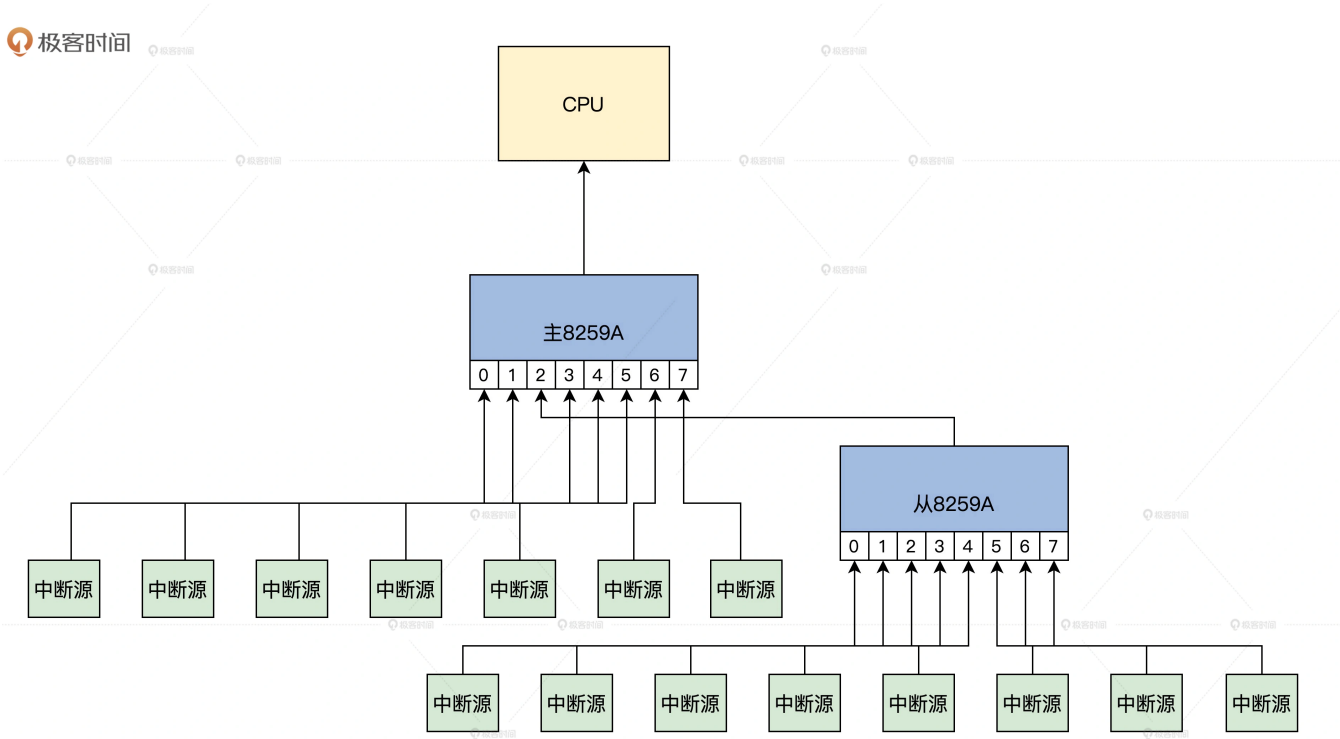
初始化中断控制器

我们把 CPU 端的中断搞定了以后，还有设备端的中断，这个可以交给设备驱动程序，但是 CPU 和设备之间的中断控制器，还需要我们出面解决。

多个设备的中断信号线都会连接到中断控制器上，中断控制器可以决定启用或者屏蔽哪些设备的中断，还可以决定设备中断之间的优先线，所以它才叫中断控制器。

x86 平台上的中断控制器有多种，最开始是 8259A，然后是 IOAPIC，最新的是 MSI-X。为了简单的说明原理，我们选择了 8259A 中断控制器。

8259A 在任何 x86 平台上都可以使用，x86 平台使用了两片 8259A 芯片，以级联的方式存在。它拥有 15 个中断源（即可以有 15 个中断信号接入）。让我们看看 8259A 在系统上的框架图：



8259A在系统上的框架图

上面直接和 CPU 连接的是主 8259A，下面的是从 8259A，每一个 8259A 芯片都有两个 I/O 端口，我们可以通过它们对 8259A 进行编程。主 8259A 的端口地址是 0x20，0x21；从 8259A 的端口地址是 0xA0，0xA1。

下面我们来做代码初始化，我们程序员可以向 8259A 写两种命令字： ICW 和 OCW；ICW 这种命令字用来实现 8259a 芯片的初始化。而 OCW 这种命令用来向 8259A 发布命令，以对其进行控制。OCW 可以在 8259A 被初始化之后的任何时候被使用。

我已经把代码定好了，放在了 8259.c 文件中，如下所示：

复制代码

```
void init_i8259()
{
    //初始化主从8259a
    out_u8_p(ZIOPT, ICW1);
    out_u8_p(SIOPT, ICW1);
    out_u8_p(ZIOPT1, ZICW2);
    out_u8_p(SIOPT1, SICW2);
```

```
    out_u8_p(ZIOPT1, ZICW3);
    out_u8_p(SIOPT1, SICW3);
    out_u8_p(ZIOPT1, ICW4);
    out_u8_p(SIOPT1, ICW4);
    //屏蔽全部中断源
    out_u8_p(ZIOPT1, 0xff);
    out_u8_p(SIOPT1, 0xff);
    return;
}

```

如果你要了解 8259A 的细节，就是上述代码中为什么要写入这些数据，你可以自己在 Intel 官方网站上搜索 8259A 的数据手册，自行查看。

这里你只要在 init_halintupt() 函数的最后，调用这个函数就行。你有没有想过，既然我们是研究操作系统不是要写硬件驱动，为什么要在初始化中断控制器后，屏蔽所有的中断源呢？因为我们 Cosmos 在初始化阶段还不能处理中断。

到此，我们的 Cosmos 的 hal 层初始化就结束了。关于内存管理器的初始化，我会在内存管理模块讲解，你先有个印象就行。

进入内核层

hal 层的初始化已经完成，按照前面的设计，我们的 Cosmos 还有内核层，我们下面就要进入到内核层，建立一个文件，写上一个函数，作为本课程的结尾。

但是这个函数是个**空函数**，目前什么也不做，它是为 Cosmos 内核层初始化而存在的，但是由于课程只进行到这里，所以我只是写个空函数，为后面的课程做好准备。

由于内核层是从 hal 层进入的，必须在 hal_start() 函数中被调用，所以在此完成这个函数——init_krl()。

复制代码

```
void init_krl()
{
    //禁止函数返回
    die(0);
    return;
}

```

下面我们在 hal_start() 函数中调用它就行了，如下所示

复制代码

```
void hal_start()
{
    //初始化Cosmos的hal层
    init_hal();
    //初始化Cosmos的内核层
    init_krl();
    return;
}
```

从上面的代码中，不难发现 Cosmos 的 hal 层初始化完成后，就自动进入了 Cosmos 内核层的初始化。至此本课程已经结束。

重点回顾

写一个 C 函数是容易的，但是写操作系统的第一个 C 函数并不容易，好在我们一路坚持，没有放弃，才取得了这个阶段性的胜利。但温故而知新，对学过的东西要学而时习之，下面我们来回顾一下本课程的重点。

1.Cosmos 的第一个 C 函数产生了，它十分简单但极其有意义，它的出现标志着 C 语言的运行环境已经完善。从此我们可以用 C 语言高效地开发操作系统了，由爬行时代进入了跑步前行的状态，可喜可贺。

2. 第一个 C 函数，干的第一件重要工作就是**调用 hal 层的初始化函数。**这个初始化函数首先初始化了平台，初始化了机器信息结构供内核的其它代码使用，还初始化了我们图形显示驱动、显示了背景图片；其次是初始化了内存管理相关的数据结构；接着初始了中断，中断处理框架是两层，所以最为复杂；最后初始化了中断控制器。

3. 当 hal 层初始化完成了，我们就进入了内核层，由于到了课程的尾声，我们先暂停在这里。

在这节课里我帮你写了很多代码，那些代码非常简单和枯燥，但是必须要有它们才可以。综合我们前面讲过的知识，我相信你有能力看懂它们。

思考题

请你梳理一下，Cosmos hal 层的函数调用关系。

欢迎你在留言区跟我交流互动，也欢迎把这节课转发给你的朋友和同事。

好，我是 LMOS，咱们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 设置工作模式与环境（下）：探查和收集信息

下一篇 14 | Linux初始化（上）：GRUB与vmlinuz的结构

精选留言 (13)

写留言

老王
2021-06-07

有了前面基础实验很快就做通了，只是实验步骤课程没有说
1.下载最新源码
git clone https://gitee.com/lmos/cosmos.git
2 进入课程的目录
cd cosmos/lesson13/Cosmos ...
展开

编辑回复: 感谢老铁的分享，特别赞。

3 11

neohope
2021-06-08

稍微整理了一下：

一、HAL层调用链
hal_start()
...
展开

作者回复: 老哥 6666

5

pedro
2021-06-07

[hal_start] --> [init_hal] --> [init_krl]
[init_hal] --> [init_halplaltform] --> [move_img2maxpadr] --> [init_halmm] --> [init_halintupt]
[init_krl] --> [die]
[init_halplaltform] --> [init_machbstart] --> [init_bdvideo]...
展开

作者回复: 铁汁牛皮

2 5

云师兄
2021-06-09

太硬了啊，有点磕牙，不过再咬两口试试😷

展开

编辑回复: 哈哈哈，舌尖上的操作系统，老铁加油！

2

然
2021-06-07

很好奇move_img2maxpadr(&kmachbsp);这个函数
功能：move_img2maxpadr(&kmachbsp);这个函数是把镜像文件搬到最大的物理地址处。
作用：我感觉是因为镜像文件是加载在0x4000000处，而空闲地址是从内核文件加载处开始计算的（0x2000000+内核大小）,随着内存的分配，空闲地址不断向上增长，迟早会...

展开

作者回复: 对，你理解的很正确

1

青玉白露
2021-07-05

干货太多，晚上去写一个博客总结一下。

展开

作者回复: 哈哈 这个可以有

吴建平
2021-07-01

代码走查出一个安全问题，下面这个函数里，如果for里一个满足条件的都没找到，那么后面校验的时候
retemp->saddr 就空指针了。
e820map_t *ret_kmaxmpadrcmpsz_e820map(machbstart_t *mbsp, u64_t mappadr, u64_t cpsz)...

展开

作者回复: 对的 厉害了 找出了BUG

二三子也
2021-06-18

../hal/x86/kernel.asm:6: fatal: unable to open include file `kernel.inc'
看了同学们的解决办法。这里在说一个解决办法。修改Makefile要优雅一点。
修改script/hal.S文件
将HEADFILE_PATH 这个变量的值-I ../include/halinc 修改为-I ../include/halinc/即可。

展开

作者回复: 你为什么 不用我的gitee上的代码呢

的是
2021-06-08

macos Bigsur :make all
CC -[M] 正在生成makefile... ../script/krnlobjs.S
CC -[M] 正在生成makefile... ../script/cosmoslink.S
CC -[M] 正在生成makefile... ../script/krnlbuidcmd.S
CC -[M] 正在生成makefile... ../script/krnlbuidrule.S...

展开

作者回复: 尽量使用linux

3

书剑飘零
2021-06-07

日拱一卒-Day04
有种本科时学8051，lpc1114的感觉，第一遍还是搞清楚流程，不要在乎细节，先建立一个整体的框架。

展开

编辑回复: 先抓主干，后看细节，你的学习方法很好。

springXu
2021-06-07

这课的信息量很大。有些意犹未尽的味道。

展开

作者回复: 哈哈，后面的更大

aagu
2021-06-07

函数名和结构体字段这种简写命名看得很迷糊，还得要加注释补充说明，老师能说一下这样做的用意吗？

作者回复: 是你自己还没有习惯

Victor
2021-06-07

在ubuntu 18.04环境下make时报错：
root@ubuntu1804:~/LMOS/cosmos/lesson13/Cosmos# make all
Initldr:清理全部已构建文件... ^_^
**********正在开始编译构建系统**********
AS -[M] 正在构建... ../ldrkrl/imginithead.asm...

展开

作者回复: 是不是下载 的完整代码

1